Kotlin - Inheritance

Inheritance can be defined as the process where one class acquires the members (methods and properties) of another class. With the use of **inheritance** the information is made manageable in a hierarchical order.

A class which inherits the members of other class is known as **subclass** (derived class or child class) and the class whose members are being inherited is known as **superclass** (base class or parent class).

Inheritance is one of the key features of object-oriented programming which allows user to create a new class from an existing class. Inheritance we can inherit all the features from the base class and can have additional features of its own as well.

All classes in Kotlin have a common superclass called **Any**, which is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

Kotlin superclass **Any** has three methods: **equals()**, **hashCode()**, and **toString()**. Thus, these methods are defined for all Kotlin classes.

Everything in Kotlin is by default **final**, hence, we need to use the keyword **open** in front of the class declaration to make it inheritable for other classes. Kotlin uses operator **":"** to inherit a class.

Example:

Take a look at the following example of inheritance.

```
open class ABC {
   fun think () {
      println("Hey!! i am thiking ")
   }
}
class BCD: ABC(){ // inheritence happend using default constructor
}

fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

When you run the above Kotlin program, it will generate the following output:

```
Hey!! i am thiking
```

Overriding Methods

Now, what if we want to **override** the think() method in the child class. Then, we need to consider the following example where we are creating two classes and override one of its function into the child class.

```
open class ABC {
   open fun think () {
      println("Hey!! i am thinking ")
   }
}
class BCD: ABC() { // inheritance happens using default constructor
   override fun think() {
      println("I am from Child")
   }
}
fun main(args: Array<String>) {
   var  a = BCD()
   a.think()
}
```

When you run the above Kotlin program, it will generate the following output:

```
I am from Child
```

A member marked with keyword **override** is itself **open**, so it may be overridden in subclasses. If you want to prohibit re-overriding it then you must make it **final** as follows:

```
class BCD: ABC() {
   final override fun think() {
      println("I am from Child")
   }
}
```

Overriding Properties

The overriding mechanism works on properties in the same way that it does on methods. Properties declared on a superclass that are then redeclared on a derived class must be prefaced with the keyword **override**, and they must have a compatible type.

```
open class ABC {
   open val count: Int = 0

   open fun think () {
      println("Hey!! i am thinking ")
   }
}
```

```
class BCD: ABC() {
   override val count: Int

   init{
      count = 100
   }

   override fun think() {
      println("I am from Child")
   }

   fun displayCount(){
      println("Count value is $count")
   }
}
fun main(args: Array<String>) {
   var  a = BCD()
   a.displayCount()
}
```

When you run the above Kotlin program, it will generate the following output:

```
Count value is 100
```

> You can also override a **val** property with a **var** property, but not vice versa. This is allowed because a **val** property essentially declares a get method, and overriding it as a var additionally declares a set method in the derived class.

We can also can use the **override** keyword as part of the property declaration in a primary constructor. Following example makes the use of primary constructor to override **count** property, which will take default value as 400 in case we do not pass any value to the constructor:

```
open class ABC {
   open val count: Int = 0

   open fun think () {
      println("Hey!! i am thinking ")
   }
}
class BCD(override val count: Int = 400): ABC() {

   override fun think() {
      println("I am from Child")
   }

   fun displayCount(){
      println("Count value is $count")
   }
}
fun main(args: Array<String>) {
   var a = BCD(200)
   var b = BCD()
   a.displayCount()
```

```
    b.displayCount()
 }
```

When you run the above Kotlin program, it will generate the following output:

```
Count value is 200
Count value is 400
```

Derived Class Initialization Order

When we create an object of a derived class then constructor initialization starts from the base class. Which means first of all base class properties will be initialized, after that any derived class instructor will be called and same applies to any further derived classes.

This means that when the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized.

```
open class Base {
   init{
      println("I am in Base class")
   }
}
open class Child: Base() {
   init{
      println("I am in Child class")
   }
}
class GrandChild: Child() {
   init{
      println("I am in Grand Child class")
   }
}
fun main(args: Array<String>) {
   var a = GrandChild()
}
```

When you run the above Kotlin program, it will generate the following output:

```
I am in Base class
I am in Child class
I am in Grand Child class
```

Access Super Class Members

Code in a derived class can call its superclass functions and properties directly using the **super** keyword:

```
open class Base() {
   open val name:String
   init{
      name = "Base"
   }
   open fun displayName(){
      println("I am in " +  this.name)
   }
}
class Child(): Base() {
   override fun displayName(){
      super.displayName()
      println("I am in " + super.name)

   }
}
fun main(args: Array<String>) {
   var a = Child()
   a.displayName()
}
```

When you run the above Kotlin program, it will generate the following output:

```
I am in Base
I am in Base
```

Overriding rules

If a child class inherits multiple implementations of the same member from its immediate superclasses, then it must override this member and provide its own implementation.

This is different from a child class which inherits members from a single parent, in such case case it is not mandatory for the child class to provide the implementation of all the open members.

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

It's fine to inherit from both **Rectangle** and **Polygon**, but both of them have their implementations of **draw()** method, so you need to override **draw()**in Square and provide a separate implementation for it to eliminate the ambiguity.

Quiz Time (Interview & Exams Preparation)

Q 1 - Which key is used to make a class inheritable:

A - abstract

B - override

C - open

D - None of the above

Q 2 - Which statement is correct from Kotlin inheritance point of view:

A - Kotlin allows to inherit multiple classes in a child class

B - Kotlin allows to override parent class properties and methods

C - Kotlin initializes properties of the classes in sequence starting from Base class to Child Class and then to Grand Child Class.

D - All the bove

Q 3 - How we can access a Kotlin parent class member in the child class?

A - If it is open in the parent class then child class can override it and access it.

B - If it is not open then child class can access it using **super**.

C - A and B statements are correct

D - We can access Kotlin parent class members in child class without any restrictions.